

APPENDIX A

Network-Level Procedures

The notation $LSU(update_list)$ represents a link-state-update message that includes the updates (u, v, c, sn) in the $update_list$.

```

5      Process_Update(i, nbr, in_message){
        // Called when an update message in_message is received from nbr.
        Update_Topology_Table(i, nbr, in_message, update_list).
        Update_Parents(i).
        For each node src in TT_i {
10           Let update_list(src) consist of all tuples (k, l, c, sn) in update_list such that
              k = src.
              If update_list(src) is nonempty
                  Send message LSU(update_list(src)) to children_i(src).}}

Update_Topology_Table(i, nbr, in_message, update_list){
15     Set update_list to empty list.
     For each ((u,v,c,sn) in in_message) {
         If (p_i(u) == nbr) {
             If ((u,v) is in TT_i and sn > TT_i(u,v).sn) {
                 Add (u,v,c,sn) to update_list.
                 Set TT_i(u,v).sn = sn.
                 Set TT_i(u,v).c = c.
                 If (sn > sn_i(u)) Set sn_i(u) = sn.}
             If ((u,v) is not in TT_i) {
20                 Add (u,v,c,sn) to TT_i.
                 Add (u,v,c,sn) to update_list.
                 If (sn > sn_i(u)) Set sn_i(u) = sn.}}}}

Link_Change(i,j){
    // Called when the cost of link (i,j) changes.
    If (|TT_i(i,j).c - cost(i,j)|/TT_i(i,j).c > epsilon) {
30         Set TT_i(i,j).c = cost(i,j).
         Set TT_i(i,j).sn = current time stamp SN_i.
         Set update_list = {(i, j, TT_i(i, j).c, TT_i(i, j).sn)
         Send message LSU(update_list) to children_i(i).}}

Link_Down(i,j){
35     // Called when link (i,j) goes down.
     Remove j from N_i.
     Set TT_i(i,j).c = infinity.

```

```

Set TT_i(i,j).sn = current time stamp SN_i.
Update_Parents(i).
For each (node src in TT_i) remove j from children_i(src).
Set update_list = {(i,j, infinity, TT_i(i,j).sn)}.
5 Send message LSU(update_list) to children_i(i).}

Link_Up(i,j){
    // Called when link (i,j) comes up.
    Add j to N_i.
    Set TT_i(i,j).c = cost(i,j).
10 Set TT_i(i,j).sn = current time stamp SN_i.
    Update_Parents(i).
    Set update_list = {(i, j, TT_i(i,j).c, TT_i(i,j).sn)}.
    Send message LSU(update_list) to children_i(i).}

Update_Parents(i){
15 Compute_New_Parents(i)
    For each (node k in N_i){
        Set cancel_src_list(k), src_list(k), and sn_list(k) to empty.}
    For each (node src in TT_i such that src != i){
        If (new_p_i(src) != p_i(src)){
20 If (p_i(src) != NULL){
            Set k = p_i(src).
            Add src to cancel_src_list(k).}
        Set p_i(src) = new_p_i(src).
        If (new_p_i(src) != NULL){
25 Set k = new_p_i(src).
            Add src to src_list(k).
            Add sn_i(src) to sn_list(k).}}}
    For each (node k in N_i){
        If (src_list(k) is nonempty){
30 Send message NEW PARENT(src_list(k), sn_list(k)) to k.}
        If (cancel_src_list(k) is nonempty{
            Send message CANCEL PARENT(cancel_src_list(k)) to k.}}}

Compute_New_Parents(i){
    For each (node src in TT_i such that src != i){
35 Set new_p_i(src) = NULL.}
    Compute min-hop paths using Dijkstra.
    For each (node src in TT_i such that src != i){
        Set new_p_i(src) equal to the neighbor of node i along the minimum-hop
        path from i to src.}}

40 Process_New_Parent(i, nbr, src_list, sn_list){
    // Called when node i receives a NEW PARENT(src_list, sn_list) message from
    nbr.
    Set update_list to empty list.

```

```

For each (node src in src_list) {
    Let sn_list.src denote the sequence number corresponding to src in sn_list.
    Add nbr to children_i(src).
    Set new_updates = {(k, l, c, sn) in TT_i such that k = src and sn >
5      sn_list.src}.
    Add new_updates to update_list.}
    Send message LSU(update_list) to nbr.}

Process_Cancel_Parent(i,nbr,src_list){
10  // Called when node i receives a CANCEL PARENT(src_list) message from nbr.
    For each (node src in src_list) remove nbr from children_i(src).}

Send_Periodic_Updates(i){
    Set update_list to empty.
    For each (j in N_i such that TT_i(i,j). c != infinity){
15      Set TT_i(i,j).sn = current time stamp SN_i.
      Add (i, j, TT_i(i,j).c, TT_i(i,j).sn) to update_list. }
    Send message LSU(update_list) to children_i(i).}

Compute_New_Parents2(i){
    S ← ∅;
    For each (v ∈ TT_i) {
        Set d(v) = infinity;
        Set pred(v) = NULL;
        Set new_p_i(v) = NULL; }
    d(i) ← 0;
    While (there exists w ∈ TT_i – S such that d(w) < infinity){
        Set u = node w ∈ TT_i – S that minimizes d(w);
        Set S = S ∪ {u};
        For each (v such that (u, v) ∈ TT_i) {
            If (d(u) + 1 < d(v) or [d(u) + 1 = d(v) and new_p_i(u) = p_i(v)]) {
20              Set d(v) = d(u) + 1;
              Set pred(v) = u;
              If (u = i) Set new_p_i(v) = v;
              Else Set new_p_i(v) = new_p_i(u); } } } }
25
30

```

Partial-Topology 1

35 The function Mark_Special_Links() is called whenever the parent p_i(src) or the set of children children_i(src) for any source src changes. The notation LSU(update_list) represents a link-state-update message that includes the updates (u, v, c, sn, sp) in the update_list, where sp is

a single bit that indicates whether the link is “special”, i.e., whether it should be broadcast to all nodes.

```

Mark_Special_Links(i){
    For all (outgoing links (i,j)) {Set TT_i(i,j).sp = 0;}
5    For all (nodes src != i){
        if (p_i(src) != NULL and p_i(src) != src){
            Set TT_i(i, p_i(src)).sp = 1;} //Link is special.
            For all (nodes j in children_i(src)){
                Set TT_i(i,j).sp = 1;} //Link is special.
10    }
}

Update_Topology_Table(i, nbr, in_message, update_list){
    Set update_list to empty list.
    For each ((u,v,c,sn,sp) in in_message) {
15        If (p_i(u) = nbr) {
            If ((u,v) is in TT_i and sn > TT_i(u,v).sn) {
                Set TT_i(u,v).sn = sn.
                Set TT_i(u,v).c = c.
                Set TT_i(u,v).sp = sp.
                (Only links marked as special are forwarded.)
                If (sp = 1) Add (u,v,c,sn,sp) to update_list.
                If (sn > sn_i(u)) Set sn_i(u) = sn.}
20        If ((u,v) is not in TT_i) {
            Add (u,v,c,sn,sp) to TT_i.
            If (sp = 1) Add (u,v,c,sn,sp) to update_list.
            If (sn > sn_i(u)) Set sn_i(u) = sn.}}}}

Process_Update(i, nbr, in_message){
    // Called when an update message in_message is received from nbr.
30    Update_Topology_Table(i, nbr, in_message, update_list).
    Update_Parents(i).
    Mark_Special_Links(i).
    For each node src in TT_i {
        Let update_list(src) consist of all tuples (k, l, c, sn, sp) in update_list such
35        that k = src.
        If update_list(src) is nonempty
            Send message LSU(update_list(src)) to children_i(src).}}

Link_Change(i,j){
    // Called when the cost of link (i,j) changes.
40    If (|TT_i(i,j).c - cost(i,j)|/TT_i(i,j).c > epsilon) {
        Set TT_i(i,j).c = cost(i,j).
        Set TT_i(i,j).sn = current time stamp SN_i.

```

```

Set update_list = {(i, j, TT_i(i, j).c, TT_i(i, j).sn, TT_i(i, j).sp)}.
Send message LSU(update_list) to children_i(i).}

```

```

Link_Down(i, j){
    // Called when link (i, j) goes down.
    Remove j from N_i.
    Set TT_i(i, j).c = infinity.
    Set TT_i(i, j).sn = current time stamp SN_i.
    Update_Parents(i).
    For each (node src in TT_i) remove j from children_i(src).
    Mark_Special_Links(i).
    Set update_list = {(i, j, infinity, TT_i(i, j).sn, TT_i(i, j).sp)}.
    Send message LSU(update_list) to children_i(i).}

```

```

Link_Up(i, j){
    // Called when link (i, j) comes up.
    Add j to N_i.
    Set TT_i(i, j).c = cost(i, j).
    Set TT_i(i, j).sn = current time stamp SN_i.
    Update_Parents(i).
    Mark_Special_Links(i).
    Set update_list = {(i, j, TT_i(i, j).c, TT_i(i, j).sn, TT_i(i, j).sp)}.
    Send message LSU(update_list) to children_i(i).}

```

```

Update_Parents(i){
    Compute_New_Parents(i).
    For each (node k in N_i)
        Set cancel_src_list(k), src_list(k), and sn_list(k) to empty.
    For each (node src in TT_i such that src != i){
        If (new_p_i(src) != p_i(src)){
            If (p_i(src) != NULL){
                Set k = p_i(src).
                Add src to cancel_src_list(k).}
            Set p_i(src) = new_p_i(src).
            If (new_p_i(src) != NULL){
                Set k = new_p_i(src).
                Add src to src_list(k).
                Add sn_i(src) to sn_list(k).}}}
    For each (node k in N_i){
        If (src_list(k) is nonempty){
            Send message NEW PARENT(src_list(k), sn_list(k)) to k.}
        If (cancel_src_list(k) is nonempty{
            Send message CANCEL PARENT(cancel_src_list(k)) to k.}}}

```

```

Compute_New_Parents(i){
    For each (node src in TT_i such that src != i){
        Set new_p_i(src) = NULL.}

```

```

    Compute min-hop paths using Dijkstra.
    For each (node src in TT_i such that src != i){
        Set new_p_i(src) equal to the neighbor of node i along the minimum-hop
        path from i to src.}}

5   Process_New_Parent(i, nbr, src_list, sn_list){
    //Called when node i receives a NEW PARENT(src_list, sn_list) message from
    nbr.
    Set update_list to empty list.
    For each (node src in src_list) {
10      Let sn_list.src denote the sequence number corresponding to src in sn_list.
      Add nbr to children_i(src).
      If (src != i) Set TT_i(i, nbr).sp = 1. //Link to nbr is special.
      If (src = i) Set new_updates = {(src, v, c, sn, sp) in TT_i such that
        sn > sn_list.src}.
15      If (src != i) Set new_updates = {(src, v, c, sn, sp) in TT_i such that
        sn > sn_list.src and sp = 1}. //Only special links are sent.
      Add new_updates to update_list.
      Send message LSU(update_list) to nbr.}

    Process_Cancel_Parent(i,nbr,src_list){
20      // Called when node i receives a CANCEL PARENT(src_list) message from nbr.
      For each (node src in src_list) remove nbr from children_i(src).
      Mark_Special_Links(i). }

    Send_Periodic_Updates(i){
25      Set update_list to empty.
      For each (j in N_i such that TT_i(i,j).c != infinity){
        Set TT_i(i,j).sn = current time stamp SN_i.
        Add (i, j, TT_i(i,j).c, TT_i(i,j).sn, TT_i(i,j).sp) to update_list. }
      Send message LSU(update_list) to children_i(i).}

```

Partial-Topology 2

```

30   Update(i, k, in_message){
    Update_Topology_Table(i, k, in_message);
    Lex_Dijkstra; // Uses lexicographic Dijkstra to compute Ti
    Generate_Updates(i, update_list);
    if (k does not equal i and update_list is non-empty){
35      Send_Updates_Children(i, update_list);
      Update_Parents(i);
    }

    Send_Updates_Children(i, update_list){
40      For each (node k ∈ Ni) {out_message(k) ← 0;}
      For each (node src ∈ TT_i s.t. src does not equal i){
        update_list(src) ← {(k, l, c) ∈ update_list s.t. k = src};

```

```

        for each (node  $k \in \text{children}_i(\text{src})$ ){
            Add  $\text{update\_list}(\text{src})$  to  $\text{out\_message}(k)$ ;
        }
    For each (node  $k \in N_i$  s.t.  $\text{out\_message}(k)$  is non-empty){
5        Send the message  $\text{out\_message}(k)$  to node  $k$ ;
    }

Update_Topology_Table( $i, k, \text{in\_message}$ ){
    For each  $((u, v, c) \in \text{in\_message})$ {
        // Process only updates received from the parent  $p_i(u)$ 
10        if  $(p_i(u) = k \text{ or } k = i)$ {
            if  $((u, v) \notin \text{TT}_i \text{ or } c \neq \text{TT}_i(u, v).c)$ {
                 $\text{TT}_i(u, v) \leftarrow (u, v, c)$ ;
                Mark  $(u, v)$  as changed in  $\text{TT}_i$ ;
            }
        }
15    }
    if ( $\text{in\_message}$  is a PARENT_RESPONSE){
        For each ( $u$  such that  $\text{in\_message}$  includes source  $u$ ){
            if  $(p_i(u) = k \text{ and } \text{pending}_i(u) = 1)$ {
                 $\text{pending}_i(u) = 0$ ;
                For each ( $v$  such that  $\text{TT}_i$  contains an entry for  $(u, v)$ ){
                    if ( $\text{in\_message}$  does not contain update for link  $(u,$ 
20  $v)$ ){
                         $\text{TT}_i(u, v).c \leftarrow \infty$ ;
                        // indicates link should be deleted
                        Mark  $(u, v)$  as changed in  $\text{TT}_i$ ;
                    }
                }
            }
        }
25    }
}

30 }

Process_Cancel_Parent( $i, \text{nbr}, \text{src\_list}$ ){
    For each ( $\text{src} \in \text{src\_list}$ )
35         $\text{children}_i(\text{src}) \leftarrow \text{children}_i(\text{src}) - \{\text{nbr}\}$ ;
}

Generate_Updates( $i, \text{update\_list}$ ){
     $\text{update\_list} \leftarrow \emptyset$ ;
    for each (entry  $(u, v, c, c') \in \text{TT}_i$ ){
40        if  $((u, v)$  is in new  $T_i$  and  $((u, v)$  is marked as changed or is not in old  $T_i))$ {
            Add  $(u, v, c)$  to  $\text{update\_list}$ ;
             $T_i(u, v).c' \leftarrow T_i(u, v).c$ ;
             $R_i \leftarrow R_i \cup \{(u, v)\}$ ;
        }
    }
}

```

```

    }
    else if ((u, v) is in  $R_i$  but not in new  $T_i$  and  $c > c'$ ){
        Add  $(u, v, \infty)$  to  $update\_list$ ; // delete update
         $T_i(u, v).c' \leftarrow \infty$ ;
        Remove  $(u, v)$  from  $R_i$ ;
    }
    if ( $TT\_i(u, v).c = \infty$ )
        Remove  $(u, v)$  from  $TT\_i$ ;
}
}

Update_Parents(i){
    For each (node  $k \in N_i$ ){
         $cancel\_src\_list(k) \leftarrow 0$ ;
         $src\_list(k) \leftarrow 0$ ;}
    For each (node  $src \in TT\_i$  such that  $src \neq i$ ) {
         $new\_p\_i(src) \leftarrow$  next node on shortest path to  $src$ ;
        if ( $new\_p\_i(src) \neq p\_i(src)$ ){
            if ( $new\_p\_i(src) \neq NULL$ ) {
                 $k \leftarrow p\_i(src)$ ;
                 $cancel\_src\_list(k) \leftarrow cancel\_src\_list(k) \cup \{src\}$ ;
            }
            if ( $new\_p\_i(src) \neq NULL$ ){
                 $k \leftarrow new\_p\_i(src)$ ;
                 $src\_list(k) \leftarrow src\_list(k) \cup \{src\}$ ;
            }
             $p\_i(src) \leftarrow new\_p\_i(src)$ ;
        }
    }
    For each (node  $k \in N_i$ ){
        if ( $src\_list(k) \neq 0$ )
            Send NEW_PARENT( $src\_list(k)$ ) to node  $k$ ;
        if( $cancel\_src\_list(k) \neq 0$ )
            Send CANCEL_PARENT( $cancel\_src\_list(k)$ ) to node  $k$ ;
    }
}

Process_New_Parent(i, nbr,  $src\_list$ ){
     $update\_list \leftarrow 0$ ;
    for each (node  $u \in u\_list$ ) {
         $children\_i(u) \leftarrow children\_i(u) \cup \{nbr\}$ ;
         $updates(u) \leftarrow \{(u, v, c) \in TT\_i \text{ such that } (u, v) \in T_i\}$ ;
         $update\_list \leftarrow update\_list \cup updates(u)$ ;
    }
    Send PARENT_RESPONSE( $src\_list$ ,  $update\_list$ ) to  $nbr$ ;}

```